Växjö University

School of Mathematics and Systems Engineering

Reports from MSI - Rapporter från MSI

Recoder with Eclipse

Saúl Díaz González
Álvaro Pariente Alonso

June
2009

MSI
Växjö University
SE-351 95 VÄXJÖ

Report 09031
ISSN 1650-2647
ISRN VXU/MSI/DA/E/--09031/--SE

Abstract
RECODER is a Java framework aimed at source code analysis and metaprogramming. It works on several layers to offer a set of semi-automatic transformations and tools, ranging from a source code parser and unparser, offering a highly detailed syntactical model, analysis tools which are able to infer types of expressions, evaluate compile-time constants and keep cross-reference information, to transformations of the very Java sources, containing a library of common transformations and incremental analysis capabilities.

These make up an useful set of tools which can be extended to provide the basis for more advanced refactoring and metacompiler applications, in very different fields, from code beautification and simple preprocessors, stepping to software visualization and design problem detection tools to adaptive programming environments and invasive software composition.

The core system development of RECODER started in the academic field and as such, it was confined into a small platform of users. Although a powerful tool, RECODER framework lacks usability and requires extensive and careful configuration to work properly.

In order to overcome such limitations, we have taken advantage of the Eclipse Integrated Development Environment (Eclipse IDE) developed by IBM, specifically its Plugin Framework Architecture to build a tool and a vehicle where to integrate RECODER functionalities into a wide-used, well-known platform to provide a semi-automated and user-friendly interface.

In this bachelor thesis we will document how we have integrated RECODER into an Eclipse plug-in to perform a “proof of concept” transformation of an Eclipse Java Project, directly mapping the Eclipse-generated configuration of the project into RECODER's to automate the tedious task of manually configuring RECODER.

Keywords: recoder, eclipse, plug-in, integration

Contents

List of Figures

1 INTRODUCTION

In the modern programming world, Java is the top one widespread programming language, and it is been the most used one steadily since 2002[1]. This means that Sun Microsystems' creation can be found everywhere, from stand-alone desktop applications, embedded in mobile devices like phones or PDAs and even in high-demanding scientific and enterprise applications.

This results into an increasing demand for Java software in the market. However, before user demands come, software developers look for new software paradigms and more advanced tools which help them to deliver faster state-of-the-art code and technology.

In case of long-term projects which require a lot of iterations and are divided among several teams, sometimes additional advanced refactoring, transformations and analysis are required. This is where the RECODER framework comes in.

On the other hand, Eclipse Integrated Development Environment (Eclipse IDE) provides an extensible platform with a large user base (almost 3 million downloads on the Ganymede 3.4 version, which does not include GNU/Linux distributed releases[2]). It is praised for his Plug-in Architecture Framework, where additional functionalities to the IDE can be easily deployed and ported between the different platforms Eclipse IDE supports, taking advantage of the resources it provides.

1.1 Context of the thesis

The RECODER framework is a set of tools, part of the COMPOST (COMPOsition SysTem) started by Prof. Dr. Uwe Aßmann at the University of Karlsruhe (Universität Karlsruhe) in Germany, with the support from co-author Rainer Neumann. In 2001 the core system of COMPOST was released as RECODER. It provides a base of analysis and metaprogramming functionalities relating Java code.[3] Currently part of the code maintenance is carried by University of Växjö (Växjö Universitet) where several teams are devoted to optimize and update RECODER's source code.

1.2 Problem

As RECODER is a framework which requires careful and tedious configuration, it has been decided to integrate the framework into Eclipse IDE, in order to provide a user-friendly interface and a larger degree of automation on performing the code transformations.

The objective for this thesis is to develop a RECODER GUI based on the Eclipse Plug-in Framework Architecture. It will serve as a “proof of concept” for a future evolved Plug-in which will allow the full functionalities of RECODER to be supported, and maybe further extensibility.

Part of the task will require basic knowledge of RECODER workings, so we will have to research on how to configure the transformations. Currently the RECODER manual is more technical-detail oriented rather than usage-oriented.

After that knowledge of the Eclipse Rich-Client Platform (Eclipse RCP) will be needed, in short, it is necessary to know how the Eclipse Plug-in Architecture Framework works and which limitations and resources we have at our disposal.

1.3 Goal criteria

This section describes the goals we aim to reach in order to solve the problem previously described in Section 1.2. As a previous step we will need to get a good understanding on how Eclipse Plug-in Framework Architecture works, and which resources are at our disposal to design and implement the required task.

As such, we can summarize our goals into the following four points:

- **Develop a GUI for RECODER** by taking advantage of Eclipse Plug-in Framework Architecture (also known as Eclipse Rich-Client Platform)
- **Provide a user-friendly interface** which eases RECODER configuration.
- The plug-in must be able to **read classpath information and library dependencies from the Eclipse source project** in order to provide an input data set.
- **A short technical and end-user documentation** is to be provided.

1.4 Motivation

Eclipse Plug-in Architecture Framework (Eclipse Rich-Client Platform) offers a platform where to develop rapid application deployment. The framework delivers enough functionality to provide a friendly interface to the user, while keeping inner logic totally transparent.

Thanks to the widespread of the Eclipse platform, you can deploy the RECODER directly to the source of the Java project, being able to perform complex behaviour thanks to the Eclipse API, which allows to pick the project data directly, instead of solving the dependencies manually.

1.5 Outline

The structure of this paper thesis is as follows:

- Chapter 2 describes the background behind RECODER and Eclipse Rich-Client Platform, giving an overview of those technologies.
- Chapter 3 deals with the RECODER plug-in itself, explaining the approach taken to its implementation, and the solution to the problems faced to it.
- Chapter 4 gives a more technical view of the plug-in.
- Chapter 5 concludes the thesis and describes future work on the plug-in.

2 Background

This section presents a brief description of the RECODER framework and the Eclipse Rich-Client Platform which are referenced through the thesis, and also explains the necessary concepts to understand properly the platform where the RECODER plug-in is being developed into. First of all, we will describe RECODER with a more detail, then a brief overview of the Eclipse Rich-Client Platform (RCP) and by extension, each of the components which are part of it.

2.1 RECODER framework

RECODER is a Java framework that provides many kinds of analyses and transformation tools. Its development started as a part of the COMPOST following Prof. Dr. Uwe Aßmann's idea. The core system, which in 2001 would be released as RECODER, was implemented by Andreas Ludwig as part of his PhD thesis[4], with the help of Prof. Dr. Aßmann and co-author Rainer Neumann at the Software Engineering and Compiler Construction Group leaded by Prof. Dr. Goos at the University of Karlsruhe.

It provides several features, in different layers:

- **Syntactic level:** RECODER is able to perform parsing and unparsing of Java files, supported by a highly-accurate model, being able to transform while trying to preserve the comments and formatting information.
- **Semantic level:** RECODER can infer types of expressions and resolve references, even keeping cross-reference information.
- **Source level:** RECODER contains a library of analyses, code snippet generators and source code transformations.
- **Model level:** Source level transformations will eventually change the underlying program model. RECODER can analyse change impacts to the global model and perform updates automatically.

One of the limitations of the RECODER framework is its complicated command line structure and configuration for performing some of its advanced features.

2.2 Eclipse Rich-Client Platform

In most of the applications, specially on integrated development environments, all the functionalities are hard coded into the sources. However, Eclipse IDE makes use of a plug-in mechanism to provide all of its functionality on the top of the runtime system. The runtime system of Eclipse is based on Equinox, a OSGi standard compliant implementation (see Section 2.6).

This plug-in mechanism is in fact a lightweight software component framework. Java and CVS are provided by the Eclipse SDK, but additional support can be added, possibly coding different extensions in Java, C, Python and more. They can subsequently support on different libraries available for each programming language, providing Eclipse with new functionalities, ranging from typesettings to additional language support and networking and database embedded applications, like Web Services support.[5]

The key to this cohesive integration relies on the architecture. Everything in Eclipse IDE is bundled as a plug-in, with the exception of a small run-time kernel which loads

the Framework and all the available plug-ins. In that sense, all the plug-ins are created equal, and they entangle with Eclipse IDE by using different extension points, from toolbars, pop-up menus to windows and perspectives which heavily modifies the environment itself. Eclipse Foundation provides a vast, but basic array of plug-ins, and gives support to both free and commercial models.

As seen in figure 2.1, the Eclipse Framework Architecture can be divided in three families of components which work in tandem to provide an application layer to the plug-ins in order to produce a working functionality. They will be explained in further sections.

From a bottom – top approach, the lower component family is the runtime-related one, composed of the core functionalities of Eclipse and the Equinox OSGi framework implementation.

The second one provides the internal integration of the plug-in by taking advantage of Java Development Tools for Eclipse (JDT), which allow the plug-in to access Eclipse resources, as well as the possible external libraries the user can add to a plug-in as a base for bringing a new functionality into the Eclipse IDE.

The last one is user-interface oriented set of graphical tools for providing a standard look and feel to the developed plug-ins. This family is made up of the Standard Widget Toolkit (SWT), a graphical toolkit developed by Eclipse Software Foundation in which the JFace and Eclipse Workbench extensions lean on to provide further complex graphical objects and paradigms to the graphical development.

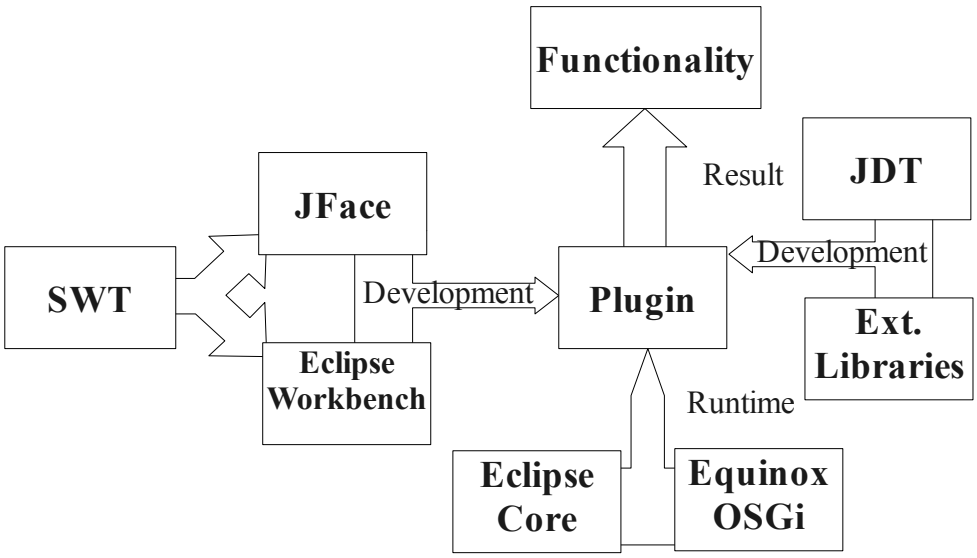

Fig. 2.1: Overview of Eclipse Framework Architecture

2.3 Standard Widget Toolkit (SWT)

The Standard Widget Toolkit (SWT) is a graphical toolkit for the Java platform. It was developed by IBM and now is maintained by Eclipse Foundation in tandem with Eclipse IDE. As such, it is the chosen graphical and widget toolkit for the user interface layer of Eclipse RCP.[6] We understand "widget" as a graphical object which displays information for the user inside a user interface, or allow him to interact with it in some way, as a button or a text area.

SWT is not a graphical toolkit per se, but a wrapper around operating system's native graphical objects. This allows the user interfaces to be portable and keep a look & feel similar to the platform they are executed on. In that sense has more similarities with the Abstract Window Toolkit (AWT), the first graphical toolkit for Java. On the other hand, Swing, the other does not rely as much on the operating system, and it offers a common look and feel which is independent on all the platforms, as it is due to of being based into Java SDK instead of relying in native widgets.[7]

The key of SWT architecture lies in the fact that it uses native widgets thanks to wrapping them on Java code. This helps on performance where in some test cases outperforms Swing's one. However, the drawback is that for each platform deployed, the interface needs to be ported every time SWT is released into a new platform. As SWT is not part of the Java release, SWT is not available for all the platforms Java is, and platform-dependant libraries must be deployed together with different versions of the application. As SWT is just an extra layer that communicates the native interface with Java, it is prone to be affected by platform-specific bugs.[8] However, as part of the Eclipse RCP, no additional requirements will be needed for development, as it guarantees portability and extensibility through all platforms supported by Eclipse IDE. In Figure, 2.2, a compared overview of the three graphical toolkit architectures is shown.

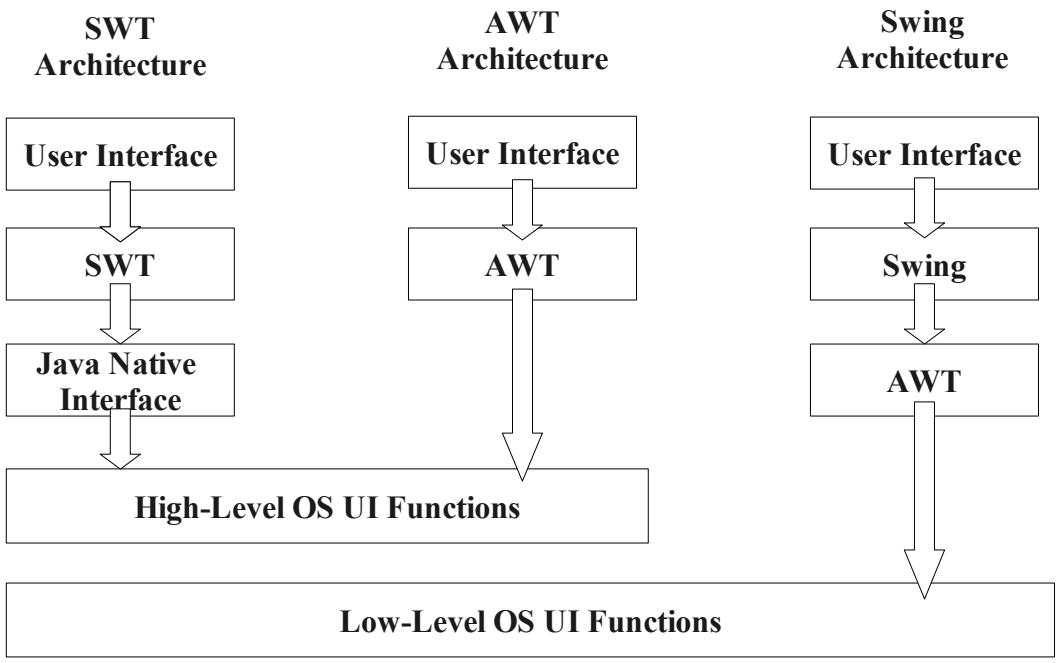

Fig. 2.2: Comparison between SWT, AWT and Swing architectures.

2.4 JFace

SWT does not provide a Model-View-Controller (MVC) architecture, the architectural pattern in which the data ("model") is decoupled from the actual way it is displayed (the "view") and the logic which processes it (the "controller"). This is provided by the JFace library, which offers a platform-independent MVC abstraction on top of SWT, providing more advanced and complex elements than the original toolkit does, like tree

views, tables and structured elements to provide data presentations.

It is composed of several packages, but the most relevant are:

- **`jface.action`** provides an interface to commonly shared UI resources as menu, status and tool bars.
- **`jface.dialogs`** provide dialog support.
- **`jface.resource`** provide support for managing resources such as fonts or images.
- **`jface.text`** provides resources for text files IO handling.
- **`jface.viewer`** provides a framework for model-based viewers, content adapters for SWT widgets.
- **`jface.windows`** provides support for window handling.
- **`jface.wizard`** provides support frameworks for wizards.

JFace is completely dependent on SWT, however SWT is not on JFace. The Eclipse Workbench is built both on SWT and JFace.[9][10]

2.5 Eclipse Workbench

The Eclipse Workbench is a set of tools, menus, editors, views and perspectives, where all the jobs will be carried out. It aims to integrate a set of tools provided by the different plug-ins installed in the system. It is in fact, the development environment, and in any given moment, one or more Workbench windows can be opened out. Figure 2.3 illustrates the Java development perspective of Eclipse IDE.

Each Workbench window contains one or more perspectives. A perspective defines the disposition of the views and the general layout of them. Each perspective provides a set of functionalities related to a specific aspect of the development, working with the same type of resources or accomplish a specific task. However, only one perspective can be activated at a time.

A perspective control manages the content of menus and toolbars. These controls are fully customizable by the user or by the developer, defining action sets the perspective is able to handle.

The views within the perspective have the aim to provide interaction with the information contained in it. They can offer alternative presentations to the data the user is handling.

The editors can be defined as a special view specialized in showing and modifying data. Usually one perspective is comprised of an editor and several views surrounding it.[11]

All these components are advanced widgets which perform complex behaviours. These are based on SWT and JFace components, and as the rest of the Eclipse RCP are fully customizable and expendable, so they can show almost anything the developer thinks of.

A plug-in can consist up to several views and perspectives which takes charge of a certain functionality. An example of this is, for example, the C/C++ development plug-in, which changes drastically the Java editor, adding Makefile editor support and several properties pages regarding C/C++ development.

All the Workbench graphical resources can be found under the packages labeled as **`org.eclipse.ui.*`**.

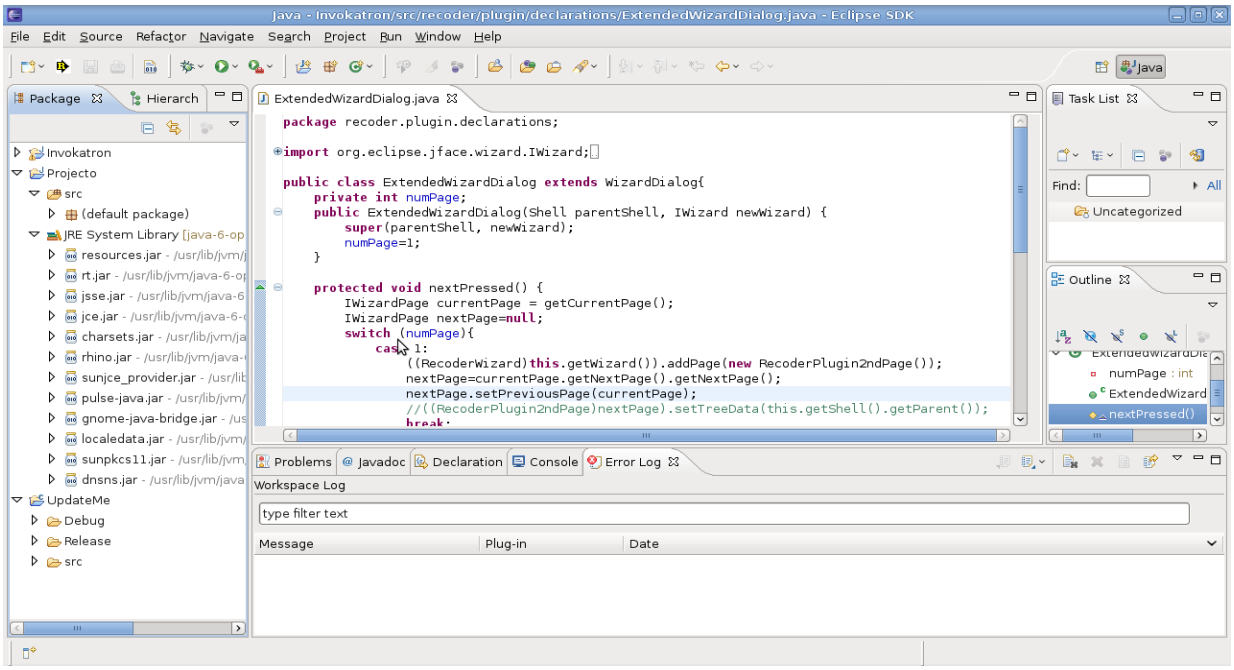

Fig. 2.3: An overview of Java development perspective in Eclipse IDE 3.4

2.6 Equinox OSGi

The Open Service Gateway Initiative (OSGi) is a consortium founded by IBM, Ericsson, Oracle and Sun Microsystems. It is also known as Dynamic Module System for Java. This platform defines the framework for deploying fully modular applications. Equinox is the implementation of the OSGi R4 core specification directly into Eclipse runtime.[12]

The point of the OSGi platform is based on deploying the application as a set of services which are able to interact between them, rather that as a whole, stand-alone program. These services can be dynamically discovered at runtime and bound to other parts of the application. Since these services are bound dynamically, these parts of the system can be stopped, removed and updated in runtime without taking offline the whole application.

In that sense, Eclipse RCP consist of a series of functionalities that are implemented as a set of plug-ins, by reusing parts of the already defined core services and framework. This plug-ins comes as a “bundle”. A bundle is a highly-coupled set of collection, classes, libraries and configuration files that can be dynamically loaded. They must explicitly declare their external dependencies.

The OSGi framework comes divided in several overlapping layers, as shown in Figure 2.4:

- **Bundles:** These are small Java applications with additional explicit manifest files
- **Services:** This layer connects the bundles by offering a publish-discover-bind listeners for Java objects.
- **Registry:** The API for managing services, providing registration, service tracking and references to the service.
- **Life-Cycle:** The API for bundle managing, providing execution services like

install, uninstall, start, stop and update.

- **Modules:** This layer defines the encapsulation level and dependency behaviour of the different bundles, in the sense of how an application manages its input and output data.
- **Security:** This module wraps all the previous ones by limiting bundle access to the predefined functionalities.
- **Execution Environment:** Defines which methods are available for a specific platform. This is a pluggable interface, as Java is constantly evolving. Currently it supports the most popular implementations of Java Virtual Machines. [13]

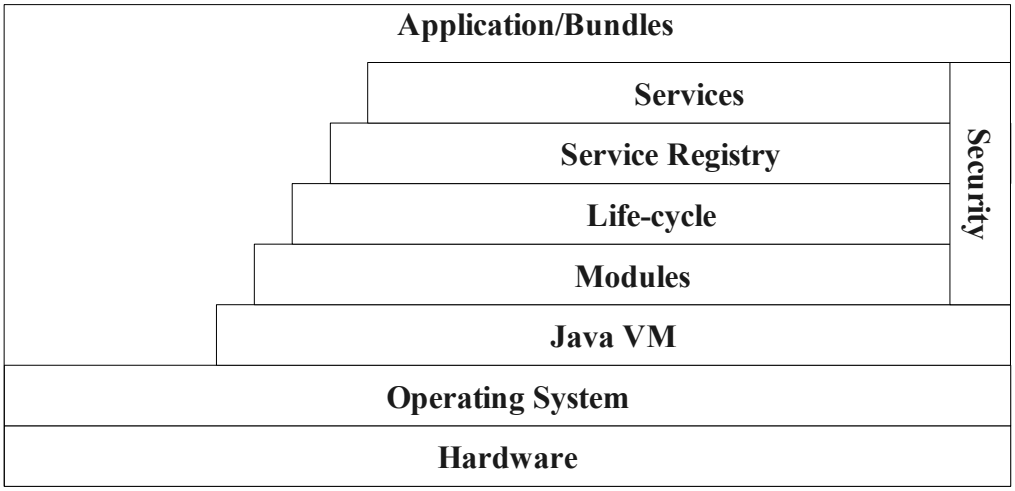

Fig. 2.4: Description of an OSGi-compliant platform.

2.7 Java Development Tools (JDT)

Java Development Tools (JDT) is a set of tool plug-ins which provides support to Java development, including Eclipse plug-ins. It includes perspectives, views and editors to add to an Eclipse Workbench regarding Java development, as many code beautification, refactoring tools and wizards.

The components of the JDT are able to act on its own, and are also entangled to provide a full development perspective or as a pluggable module for third-party applications. The list of components is as follows:

- `jdt.apt:` Adds annotation support for Java 5 projects in Eclipse.
- `jdt.core:` Contains the non UI infrastructure mostly regarding the transformation and managing of Java code and structures.
- `jdt.debug:` Provides the execution environment of the JDT, being able to communicate to any JDPA-compliant (Java Debugger Platform Architecture) Java VM, and launch Run or Debug modes.
- `jdt.text:` Offers non-invasive source code services like syntax highlight, documentation assist and code formatting.
- `jdt.ui:` Implements specific Workbench contributions like Eclipse Package Explorer, hierarchy views and it is able to locate and update references to packages, types, classes, methods and fields.[14][15]

2.8 Basics of Plug-in Development in Eclipse

In order to develop a plug-in into the Eclipse Rich Client Application, we can directly define a plug-in project inside the IDE itself. It is a special project type which bundles some additional information into a XML file called “plugin.xml”. This will serve as a cornerstone for the rest of the development.

That the plug-in is included in what we would call a “subset of Java projects”, means that we can treat it as a standard Eclipse Java project. Here is where the flexibility of the Eclipse RCP comes into play: we can easily attach an external library to the project, providing in first place its functionalities to the IDE on a real easy way. Any .jar-bundled library or API can be plugged into a plug-in, being able to add almost any functionality desired to the Eclipse IDE.

Most of the plug-in related inner configuration is made through Eclipse RCP's “plugin.xml” editor. While most of the actual coding will happen on a standard Java editor, the way that code entangles with the application itself is described editing manually plugin.xml or interacting with the editor itself. The plug-in is fully configurable and the editor will reflect the changes made to the XML and vice versa, so the developer can use different levels of depth when defining his plug-in features.

The plugin.xml is divided into several tabs, as shown in figure 2.5 and described as follows:

- **Overview:** Contains the general description of the plug-in ,allowing different basic options to be edited and provides links to several more advances features of the editor. It also contains basic instructions on the plug-in's basic operations.
- **Dependencies:** This tab describes which internal libraries does the plug-in depend on. It also provides several dependency-related tools, like a circular-dependency, unused and plug-in dependencies scanner.
- **Runtime:** In this tab the external dependencies (i.e. RECODER library) are declared. This tab also allows the user to define visibility constraints to other plug-ins, not allowing other attached systems to make use of the resources available for the plug-in.
- **Extensions:** Here are specified which parts of the graphical interface will be extended when the plug-in will be installed. The RCP allows to define new buttons on tool bars, menus and sub-menus, even new perspectives, views and editors.
- **Extension points:** The extension points define the outside interfaces our plug-in provide. In Eclipse RCP the plug-ins are not only thought to be stand-alone sets of functionalities. They can interact between each other, combining themselves to provide highly-sophisticated behaviours.
- **Build:** This tab allows the user to exclude some parts of the project from the binary build (.jar file) and/or the source build (source code). In that sense, a developer could add proprietary code which does not want to reveal on the source build of its plug-in.
- **Plugin.xml:** This tab is the actual editor of the XML files. It is a simple XMl editor. The changes made to the XML, if correct, will be reflected on the advanced editor and vice versa.
- **Build.properties**: It is a special text editor for editing the building configurations.

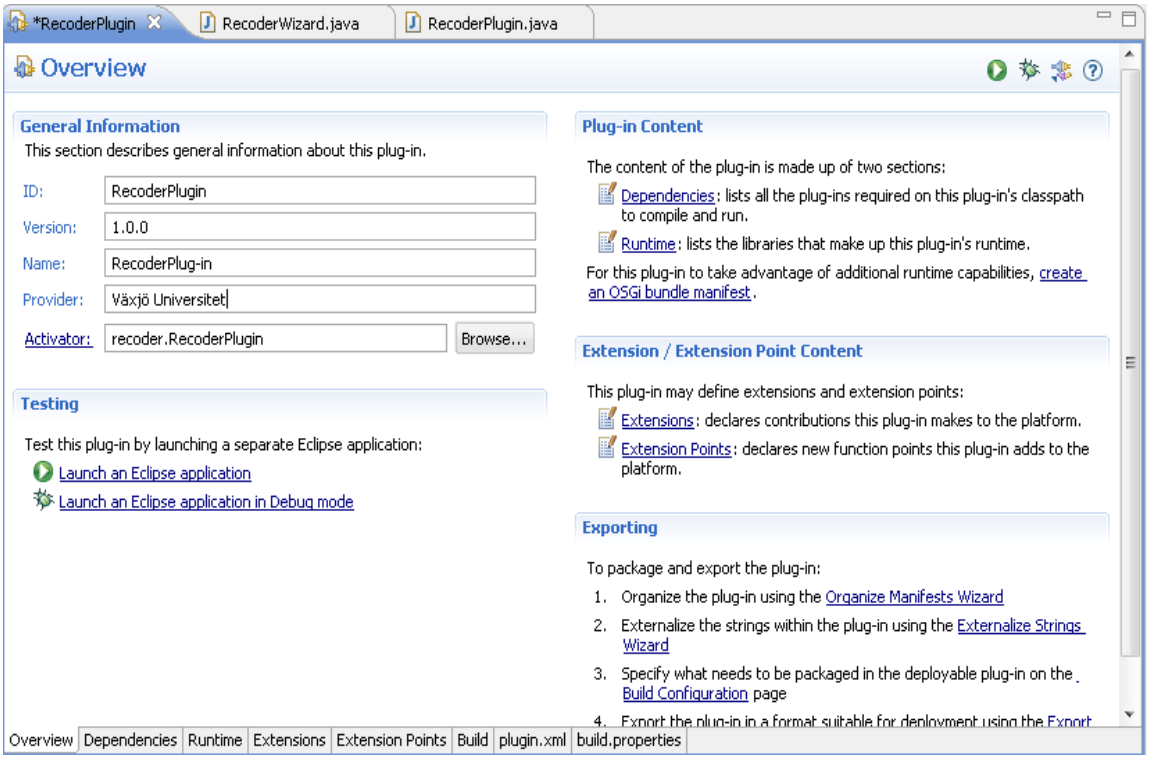

Fig. 2.5: Plugin.xml editor.

2.9 Building and installing the plug-in

We have described how to integrate the plug-in with Eclipse interface and the actual design and coding processes. This means we have a lot of related graphical, logical and meta resources ready for deployment, but we have to perform the actual building of the plug-in.

A plug-in is bundled as a special .jar file which contains the compiled sources, optionally the uncompiled sources, the plugin.xml generated during our development and the resources needed to properly work (help files, icons, etc...).

In order to deploy a version of the plug-in is necessary to have the source code of it loaded into Eclipse workspace. Under “File...” menu or by right-clicking on the Java Navigator view the “Export...” command will be available. A tree of different exporting options will appear. We will be interested on “Plug-in development” section. An example of this is shown in Figure 2.6.

From the three options, to deploy a plug-in we require to click on “Deployable plug-ins and fragments”. This will lead us to the next part of the Deployment Wizard. In this page the available deployable plug-in projects will appear on a list. A developer can choose to deploy a single, stand-alone plugin, or bundle several in a .zip file or a directory.

Also, there are additional options available like:

- Including uncompiled source files.
- Bundling all the plug-ins projects into individual .jar files for modularity.
- Saving the ANT script for further generation.
- Sign the jar file.

As stated earlier this will generate the .zip file or the directory as chosen. Inside of any of them a .jar file with the name of the plug-in project and its version. This jar file can be copied to any Eclipse release, independently of the platform, to gain access to, in this case, RECODER transformation functionalities.

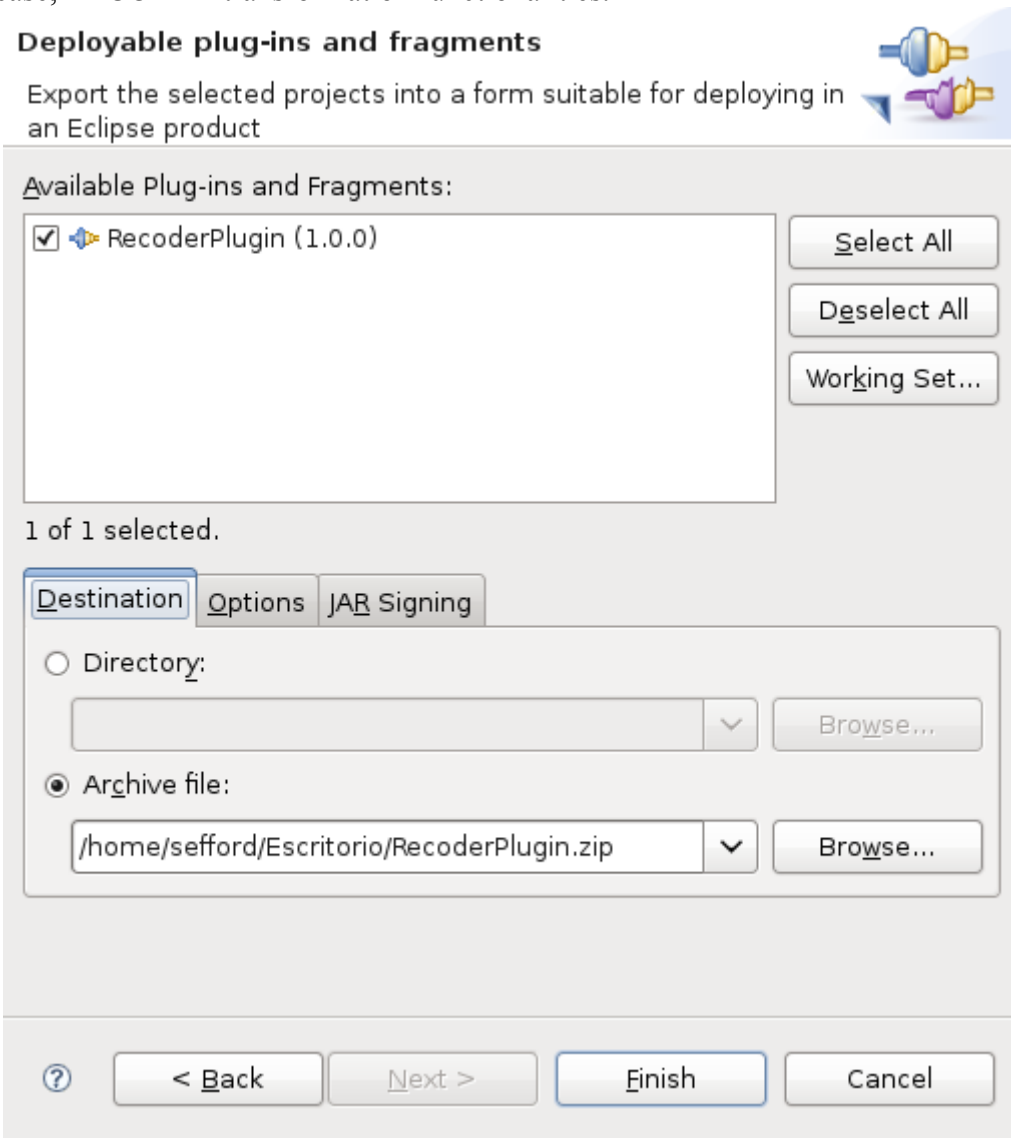

Fig. 2.6: Screenshot of the Plug-in Deployment Wizard.

3 Implementation

In this chapter we will talk about the approach we took on integrating RECODER into Eclipse IDE. First we will review the thesis objective on a general outline, based on the information we gathered from the previous chapter. Secondly we will explain in more detail how a plug-in is actually developed under Eclipse RCP, and the tools available for the process. Next, the different aspects of the development will be described in detail, including design issues, graphical integration with Eclipse and RECODER integration with the plug-in.

3.1 Approach: Integrating RECODER

The idea behind the RECODER integration is to provide a simple, easy to use interface to the framework, merging it into Eclipse IDE Java perspective. The bulk of RECODER configuration consists of defining the classpaths and library dependencies of the project, and that is the part we want to ease as much as possible.

Thus, we are substituting the generic Java project by a configured Eclipse-managed Java Project, this is, a set of Java sources with additional information about it retrieved or generated by Eclipse IDE itself. By using JDT core API we can freely access and manage the underlying Eclipse interface, getting access to the resources this provides.

First of all, the path needs to be extracted from the user input project selection. Then, RECODER will be configured calling its execution method and specifying the right parameters such as the input path extracted and worked before. Meanwhile, exclude files are taken into consideration to avoid their transformation. Finally, transformations executed by RECODER method will be stored in the output folder specified by the user. This is shown in the figure 3.1 as follows:

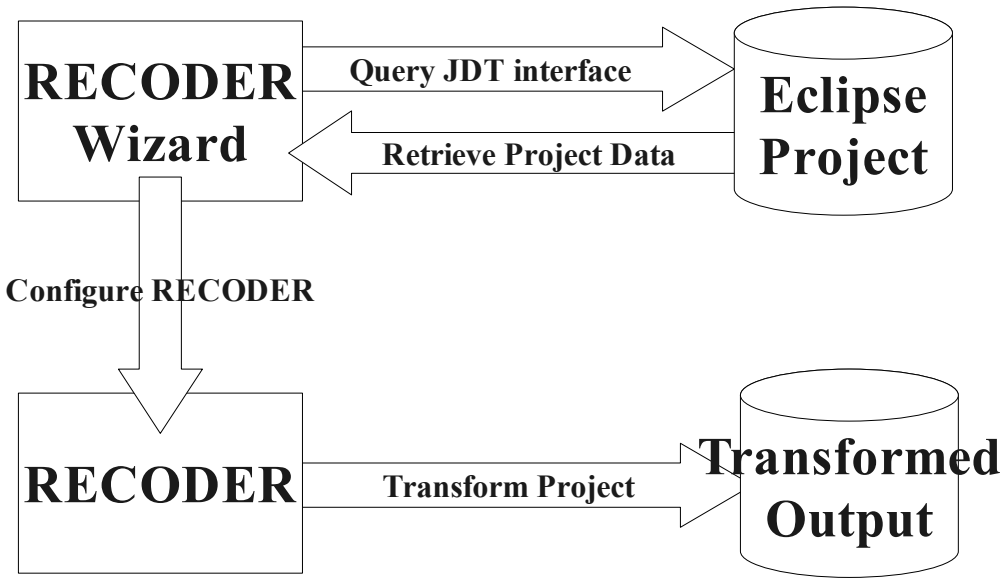

Fig. 3.1: Overview of the thesis objective

3.2 Graphical Integration with Eclipse

In order to graphically integrate RECODER we have decided to take the approach of building a easy, intuitive Wizard which will be executed from a button in the tool bar. It was previously experimented with other different approaches, like a drop down menu addition into the Navigator view or addition to the top menu bar. However, we decided that in terms of visibility and accessibility the chosen way is the best, as both menus are extensions of the tool bars and as such, they provide extra functionalities which obscures the user's sight, and tool bar provides also a visual aid the user can easily recognize, even integrating with it with Eclipse's look and feel.

Defining the new Extension to the plug-in requires several steps. First of all it is required to define the extension point itself in plugin.xml. This is done as an extension of **`org.eclipse.ui.actionSets`**. This contains the elements necessary to extend the common menus, tool bars and pop-up menus common to all the perspectives.

As an element is fully customizable, being able to define the behaviour of more than ten properties, however, we will focus on the basic necessary declarations to define an extension point of the UI. The one used on the proof-of-concept is defined as:

```
<extension id="NewRecoderAction"
                name="Recoder Proof-of-concept Transformation"
                point="org.eclipse.ui.actionSets">
…
</extension>
```

As seen, the basic outlying of an extension point of the UI (not to be mistaken with Extension Points for plug-in interface) requires at least:

- **`id field:`** Unique identifier for the extension.
- **`name:`** The alternative name which will be given to the extension point. In case of menus or buttons, this is also the name which will take when shown graphically.
- **`point:`** This is the extension point of the UI. In this case we are defining an action. Other different extension points include, without being the full list:
  - Perspectives
  - Editors
  - Views
  - Drop down menus

Additional simple level configurable properties are, for instance, visibility (**visible**, default TRUE) or **description**.

After that we cannot use an existing actionSet, as it is already defined and closed, like Source or Run. We have to define our own and customize it. This again allows the edition of several properties which are not interesting to the point of the project. The basic definition of a new actionSet is like follows:

```
<actionSet id="recoder.actionSet"
                label="Recoder Actions"
                visible="true">
…
</actionSet>
```

Again the definition requires at least a unique identifier, and a label, in order to show a certain title within the UI. Although implied, the visibility tag has been enabled as true.

Now is the point to define the different actions our actionSet will provide. As this is a proof-of-concept, only one is required, however a full-functional plug-in can hold dozens of actions which can be customized to the point of appearing only if the selected item matches some kind of class. In our case:

```
<action id="recoder.wizard.RunWizardAction"
                label="New Proof-of-Concept transformation"
                menubarPath="file/new.ext"
                toolbarPath="org.eclipse.ui.workbench.file/new.ext"
                icon="icons/RecoderIcon16.GIF"
                tooltip="Starts the New Recoder Proof-of-Concept transformation"
                class="recoder.wizard.RunWizardAction">
</action>
```

As we are defining the actual action, new properties need to be defined:

- **menubarPath / toolbarPath:** An action can be inserted simultaneously into a menu and a tool bar. Eclipse IDE has a toolbar per default menu, as it is usual that many important actions able to be performed have an entry on their correspondent menu and its subsequent tool bar.
- **icon:** As we are defining an action, it will be visually nice to have some sort of visual aid to click the correspondent action in the tool bar.
- **tooltip:** Self-explanatory, the UI will allow a small description of the action to pop-up if the mouse hovers over the action itself.
- **class:** This field will trigger the class / method which will perform the desired operation.

After that we will have defined the appearance of our proof-of-concept action into the Eclipse IDE Java Perspective.

Other interesting properties not needed to be defined in our XML but useful for future development will be mentioned in the next table:[16]

enablesFor [!, ?, +, 2+, n, *]	Enables this action for [0, 0 or 1, 1 or more, 2 or more, n items only, any] items selected.
style [push, radio, toggle, pulldown]	Attribute to define the user interface style type for the action. If omitted, then it is push by default.
hoverIcon	Defines an icon for when the mouse is hovering over it.

The results of the implementation of the extension point can be seen as a result in Figure 3.2.

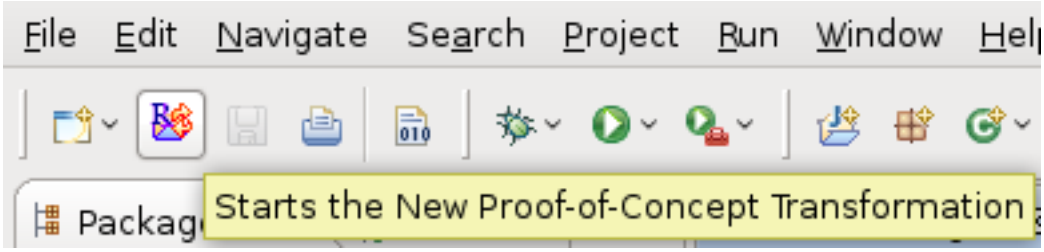

Fig. 3.2 : Screenshot of the defined action button of the plug-in.

3.3 Wizard introduction

A wizard is a help mechanism in an application that makes an easy usage to the user through different pages with the goal of performing a task.

TheWizard consists in two different pages which interact with the user specifying a new runnable configuration for RECODER in turn. On one hand, the first wizard page provides a button called 'Browse' which when clicked, gives the possibility to choose a project within workspace in order to configure the input path needs. (Figure 3.3)

On the other hand, the second wizard page shows a system tree where the user should choose a folder, in order to specify where RECODER transformations files will be stored. (Figure 3.4)

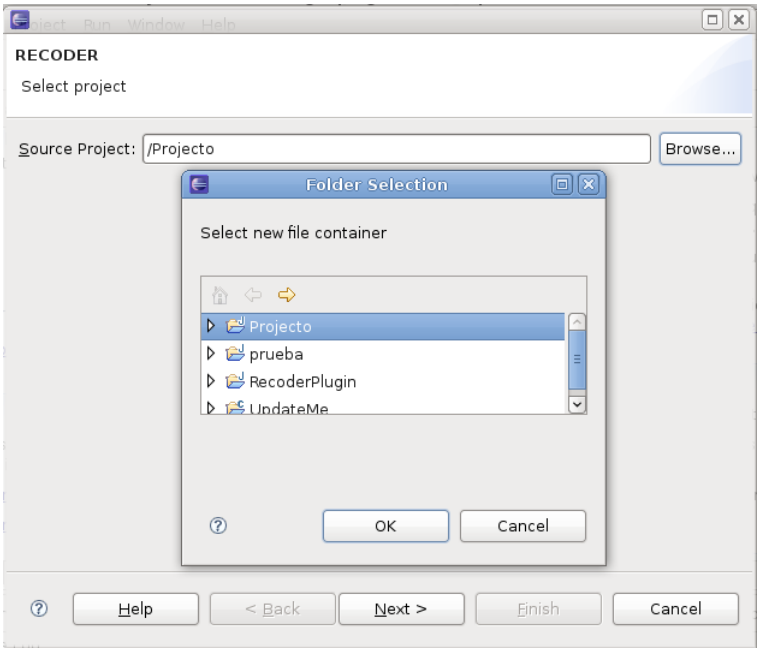

Fig. 3.3: RECODER Transformation Wizard page 1.

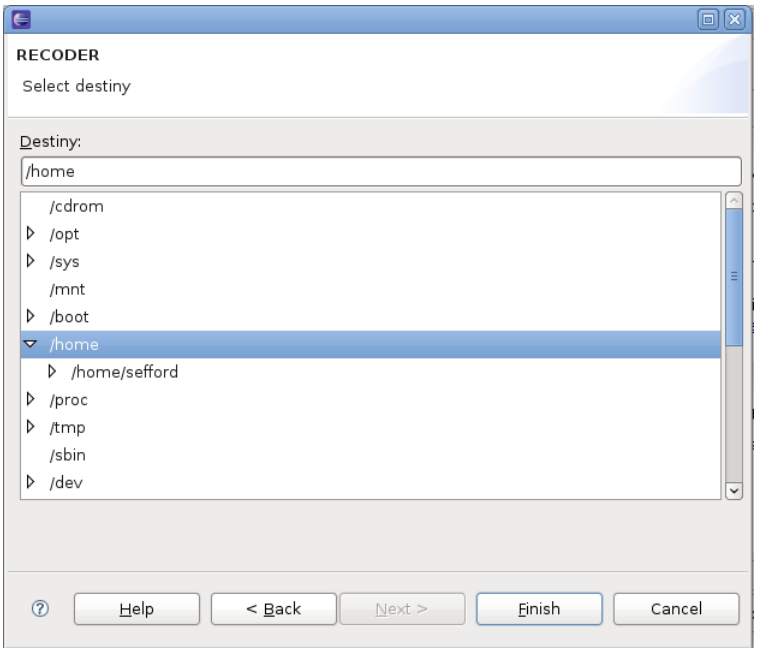

Fig. 3.4: RECODER Transformation Wizard page 2

3.4 Wizard Implementation

Wizard implementation is based upon Wizard class provided by IBM Corporation. This class handles all methods that a wizard needs to build an entirely guide graphical interface. Within these methods it is noteworthy that **`addPage`**, **`setWindowTitle`**, **`setDialogSettings`**, **`performCancel`** and **`performFinish`** are the most important ones.

The method **`performFinish`** is called when user presses the *finish* button to execute transformations. In addition, it is the responsible of all logical operations. Getting classpath configuration is one of the major task required by RECODER to specify the right input path together with the source project selected by the user. Getting the output path selected by the user and calling the execute RECODER class with the input and output path as parameters are tasks that this method performs as well.

Besides the Wizard class, it is necessary to have two classes which interface components are described. These classes are **`RecoderWizardPage`** and **`RecoderWizardPage2`**. Both classes perform different methods to handle needs required such as a browser with workspace projects or a complete system folder tree.

3.5 Wizard RECODER integration

RECODER role consists in executing the transformation with the right configuration given by the user through input and output path. A separate package has been created with a 'Test' class which makes use of configuration and execution actions of RECODER.

RECODER requires some input and output configurations to make this framework works as user needs. On one hand, an input path has to be chosen by the user in order to select which project will be transformed by RECODER. Furthermore, this input path needs to contain all resources attached to the project selected such as libraries and other projects needed for compilation. On the other hand, the user shall have the possibility to

choose an output path where all transformed code will be written to.

3.6 Applying JDT interface to extract Project Information

RECODER plug-in needs some hard configurations to specify the right input path, which is made up by different paths separated with semicolons. These paths consist in classpath configuration properties of the project selected by the user. There are three different properties that must be taken in consideration: source folders, required projects and .jar folders on the build path.

Source folders provide important relevant information about include and exclude files within the project. As projects may not be compiled, Eclipse supports one property which files or packages can be excluded from compilation. Users have the possibility to configure this property before executing RECODER plug-in in order to make a project compiles and exclude those files of transformation.

Required projects and .jar folders are necessary for the project selected by the user, to be compiled.

JDT offers some important methods to access these classpath and workspace configurations as are described in next figure.

```
IWorkspace workspace1 = ResourcesPlugin.getWorkspace();
IJavaModel javaModel = JavaCore.create(workspace1.getRoot());
IJavaProject projects[] = null;
        try {projects = javaModel.getJavaProjects();}
IClasspathEntry[] entry = projects[j].getResolvedClasspath(false);
case IclasspathEntry.CPE_SOURCE:{
        // Exclude files configuration
        exclude = entry[i].getExclusionPatterns();}
case IclasspathEntry.CPE_PROJECT:{
        // Projects attached}
case IclasspathEntry.CPE_LIBRARY:{
        //Jar libraries attached}
```

Once the user chooses the input project, the plug-in reads the classpath configuration properties and depending on which property, the path is added to the input path or some changes need to be done before. In projects and libraries related, the path is added without any change required. However, source folders need a special treatment. Excluding files can be done in different cases: excluding file by file or excluding a package which will automatically exclude files inside them.

For instance, if the user selects the project called 'myproject' and it contains one package called 'exclude', the result transformation will not take the exclude package java classes in consideration for the result. As next figure shows, exclude package changes its icon to a folder once is configured to exclude.

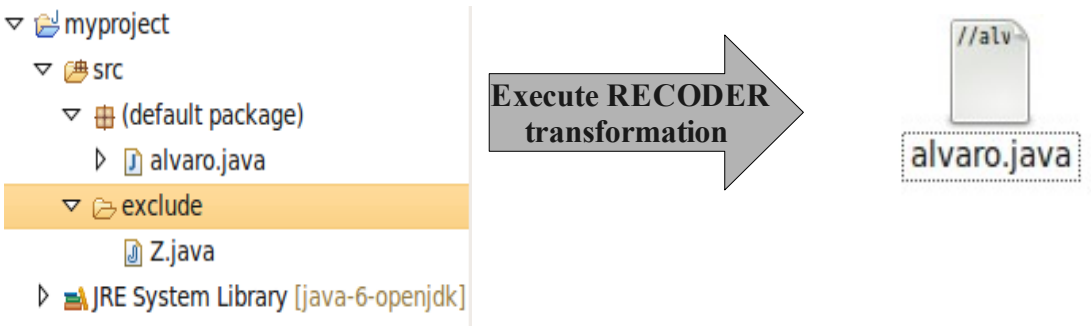

Fig. 3.5: Overview of the plug-in transforming process.

This last operation success thanks to a filter which gives the possibility to leave those exclude files out of transformation. The filter consists in a method which accepts or denies files. As a exclude List is given as a parameter while calling execute method, it forces to accept the Java classes which are not include in that list.

```
public void execute(String inputPath,String outputPath,final IPath[] exclude,
                    final List excludeFolder)
```

```
FilenameFilter filter = new FilenameFilter(){
  public boolean accept(File dir, String name) {
        List childrenExclude = null;
        for (int i=0;i<exclude.length;i++){
              if(exclude[i].toString().equals(name) || !(name.endsWith(".java")) ||
              excludeFolders.contains(name) ){
              return false;
        }
   }
  return true;
  }
};
```

Finally, RECODER takes the right input and output path calling the final execute transformation method.

```
CrossReferenceServiceConfiguration sc = new CrossReferenceServiceConfiguration();
sc.getProjectSettings().setProperty(PropertyNames.INPUT_PATH, inputPath);
sc.getProjectSettings().setProperty(PropertyNames.OUTPUT_PATH, outputPath);
new TransformAll(sc).execute();
```

4 Technical Documentation

In this section we will describe technically how the plug-in is implemented by showing its class diagram.

The core of the plug-in is, as shown in Figure 4.1, built around the RecoderWizard class. This is an extension of a JFace Wizard class (`org.eclipse.jface.wizards.Wizard`) and implementation of a new INewWizard interface (`org.eclipse.ui.INewWizard`). This is the actual RECODER wizard described in Section 3.3.

This wizard makes use of two extended WizardPages (`org.eclipse.jface.wizard.WizardPage`), RecoderWizardPage and RecoderWizardPage2 to provide the interface shown in Figures 3.3 and 3.4 respectively.

The RECODER wizard method `performFinish()` includes the actual code shown in Section 3.6 to call the `execute(string, string, Ipath[], ArrayList)` which performs the transformation, provided by `Test` class, which in turn will communicate as necessary with the RECODER library.

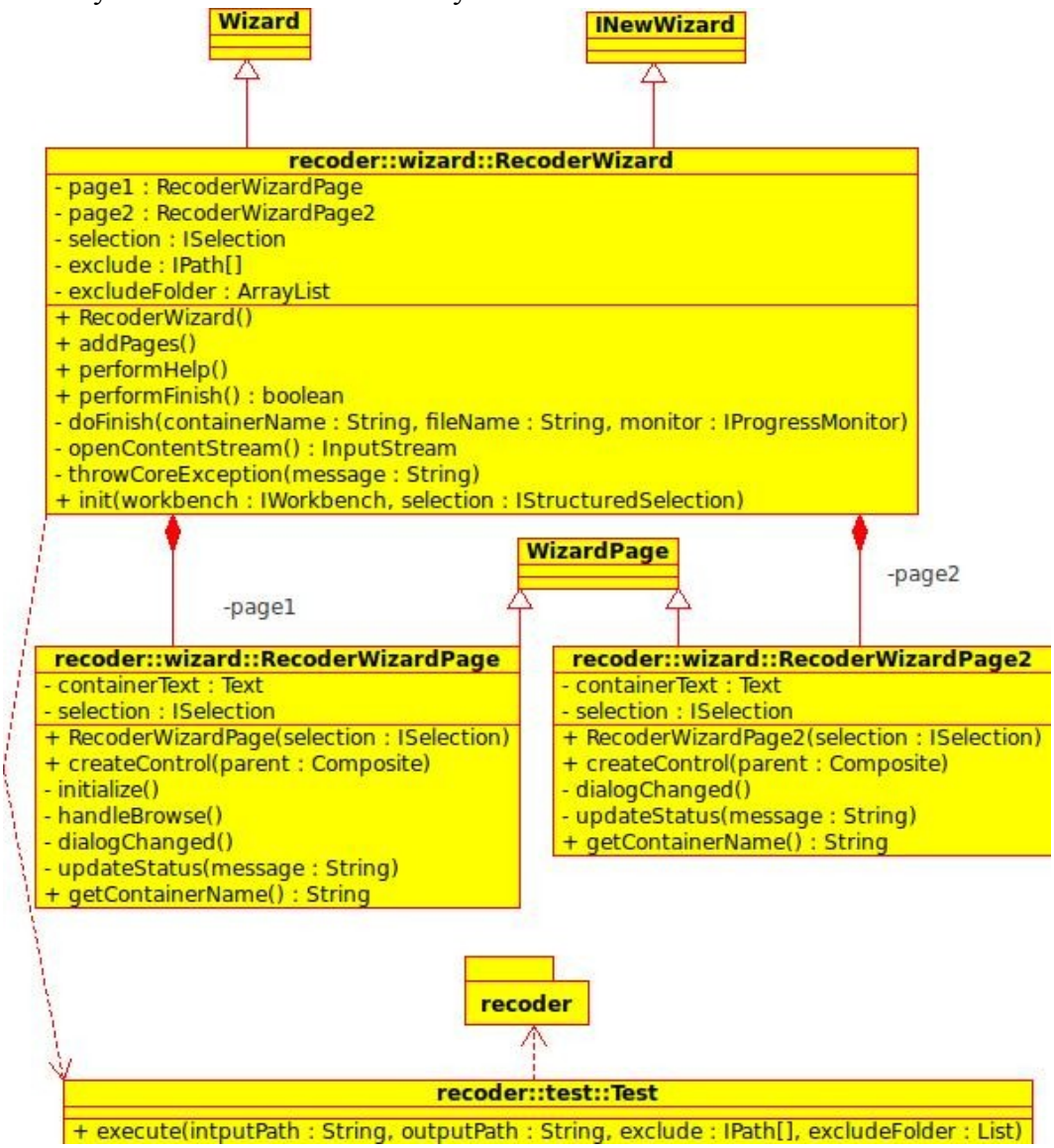

Fig. 4.1: Class diagram of the RECODER plug-in.

5 Conclusion and future work

This last chapter summarizes the results of the thesis. It will point out up to which extent the problems posed in the Section 1.3 have been solved and the criteria fulfilled. It will also further discuss the directions of future developments over the implementation.

5.1 Conclusion

As described in Section 1.3 of this thesis report, we described the problems posed to us as:

- **Develop a GUI for RECODER** by taking advantage of Eclipse Plug-in Framework Architecture (also known as Eclipse Rich-Client Platform)
- **Provide an user-friendly interface** which can ease RECODER configuration.
- The plug-in must be able to **read classpath information and library dependencies from the Eclipse source project** in order to provide an input data set.
- **A short technical and end-user documentation** which is to be provided.

As shown in Sections 3.2 and 3.3, we have developed a user interface into Eclipse. We have taken advantage of the extensibility of the Eclipse Rich-Client Platform and lean on its graphical components, JFace and Eclipse Workbench to integrate it into the Eclipse IDE platform.

We used JFace and Eclipse Workbench dependent widgets to build a proof-of-concept Wizard which allows the user to quickly map the resources inside his Eclipse-managed Java projects to the RECODER configuration by selecting a source project and an output folder, being able to perform the transformation way easier than if configuring it manually, and as such, getting rid of the nuisance of manually pointing the Java classpath and project sources, a very error-prone and sometimes confusing task. In that sense we believe we have achieved the improvement on user-friendlyness. The Wizard has been shown in Figures 3.3 and 3.4.

The plug-in also makes use of the core API of Eclipse to read the project dependencies and sources, and as as sub-task we have been able to sort out excluded sources within the project, ensuring that only user-chosen source files come into play when performing the transformation. In that sense, we have included an additional layer of configuration to RECODER, by taking advantage of some features already implemented into the IDE.

As stated on the previous chapter, and as on Appendix A, expand more the technical and end-user documentation.

We have, thus, fulfilled all our goal criteria.

5.2 Future work

The proof-of-concept implementation of RECODER plug-in serves its purpose on deliver an example of the potential of Eclipse RCP for presenting this and other many frameworks on an easier way and on a natural environment for their application.

Currently the plug-in only performs one single “proof-of-concept” transformation of the sources. The next step on development would be to integrate the full transformations library into the plug-in. In that sense RECODER could provide a more plug-in oriented

configuration interface to produce more reusable pieces of code. However, it is the plug-in duty to adapt to RECODER design and not the other way around.

In terms of design, the plug-in could offer more extensibility by some kind of additional information on which transformations are available to the plug-in, so it is adaptable to different versions of the RECODER library.

Graphically, the plug-in delivers an easy to use interface through a button placed in the tool bars. This will display the Project Transformation Wizard. However, the Wizard could have been made additionally available through specific RECODER menu extension into the menu bar, and into a drop down menu into the Java Navigator.

By using the API provided by the Java Navigator, the transformations could be applied specifically to only packages or single source files, or load the Project automatically into the Wizard instead of picking it manually from the Chooser.

It should be also desirable that, as we are working inside Eclipse IDE, the source Eclipse Java project should be transformed into a new Eclipse Java project.

References

[1] Tiobe Programming Community Index. (09/05/08)
http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[2] Eclipse Software Foundation. “Eclipse Download Page”. (09/05/09)
http://www.eclipse.org/downloads

[3] Rainer Neumann, Uwe Aßmann. “RECODER Manual”, 2008

[4] Andreas Ludwig, Automatische Transformation großer Softwaresysteme, dec 2002

[5] Eclipse Plugin Central. (09/05/09) http://www.eclipseplugincentral.com/

[6] “Why does Eclipse use SWT?”. (09/05/13)
http://wiki.eclipse.org/FAQ_Why_does_Eclipse_use_SWT%3F

[7] Manilli, Mauro. “Swing and SWT: A Tale of Two Java GUI Libraries”. (09/05/15)
http://www.developer.com/java/other/article.php/2179061

[8] Steve Northover & Carolyn MacLeod, (2001). “Creating your own Widgets using SWT”. (09/05/15) http://www.eclipse.org/articles/Article-Writing%20Your%20Own%20Widget/Writing%20Your%20Ow%20Widget.htm

[9] Eclipse Software Foundation. “The JFace UI Framework”. (09/05/16)
http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.platform.doc.isv/guide/jface.htm

[10] Eclipse Software Foundation. “Eclipse API Reference”. (09/05/18)
http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/api/overview-summary.html

[11] The Eclipse Workbench. (09/05/18) http://gild.cs.uvic.ca/student%20docs/workbench.html

[12] Eclipse Software Foundation. “Equinox”.(09/05/18)
http://www.eclipse.org/equinox/

[13] Wikipedia. “OSGi”.(09/05/18) http://en.wikipedia.org/wiki/OSGi

[14] Eclipse Software Foundation. "JDT Programmer's Guide".(09/05/19)
http://help.eclipse.org/ganymede/topic/org.eclipse.jdt.doc.isv/guide/jdt_int.htm

[15] Eclipse Software Foundation. “JDT API Reference”.(09/05/19)
http://help.eclipse.org/ganymede/topic/org.eclipse.jdt.doc.isv/reference/api/overview-summary.html

[16] Eclipse Software Foundation. “Eclipse Extension Points Reference”.(09/05/19)
http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/extension-points/index.html

Appendix A: End-User Manual

This manual explains how to install and make use of RECODER plug-in in Eclipse.

A.1 Installing RECODER plug-in into Eclipse IDE

The RECODER plug-in package consists of the Eclipse Project source code and the compiled plug-in jar. In order to install it, the user only needs to extract RecoderPlugin_[version].jar into the "plugin" sub folder inside the Eclipse IDE installation.

> **Tip:** Sometimes Eclipse's cache is not properly updated. In that case try to run "eclipse -clean" for forcing the cache cleaning.

A.2 Accessing RECODER plug-in functionality

Once the installation is complete, a new icon should be available in the tool bar. Clicking on it will provide access to the Proof-of-Concept Transformation, by opening the RECODER Transformation Wizard.

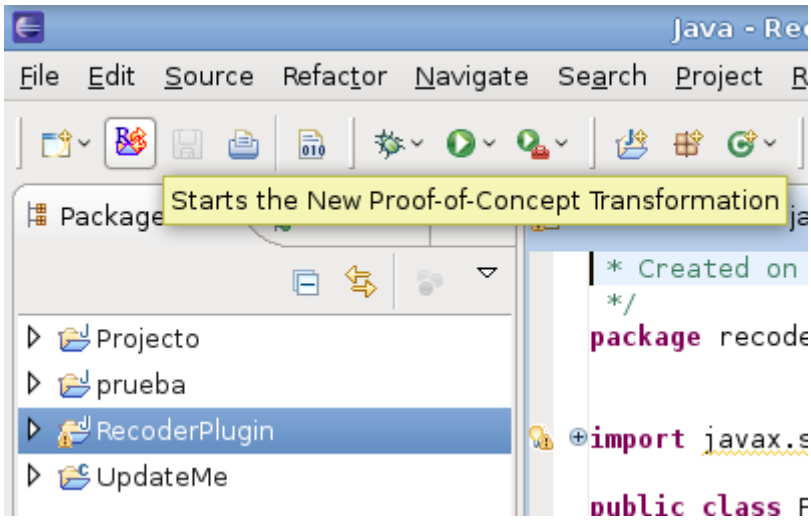

Fig. A.1: Location of RECODER plug-in transformation icon.

A.3 Select Input Path

The first page is a simple button together with a selection screen which will lead to a Folder Selection Dialog. The Folder Selection Dialog is constrained inside Eclipse's Workspace and will allow available Eclipse projects to be selected.

The user is able to cancel the transformation at any time by clicking the “Cancel” button. Once a project is selected, the “Next >” button will be available for the next step in the transformation.

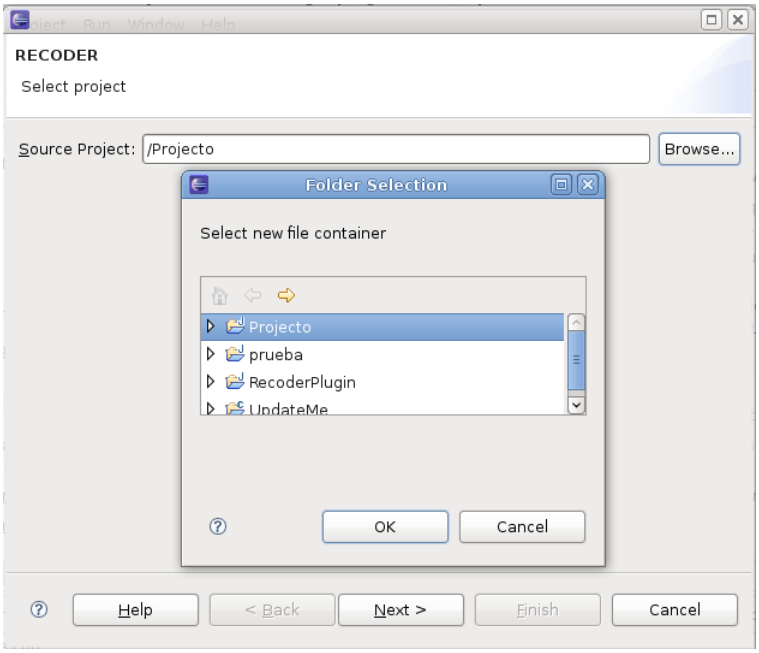

Fig. A.2: Project Selection Dialog for configuring Eclipse Source Project

A.4 Select Output Path

This second Wizard page provides a folder system tree which the user can navigate in order to choose the output folder. In this output folder, RECODER will output the result of the transformation.

Once a folder is selected, the “Finish” button will enable, allowing the plug-in to perform the transformation.

Tip: Watch out that the Project is configured properly. The plug-in can read which files are excluded from the compilation. However due to RECODER's nature, a non-compiling Project will not be able to transform.

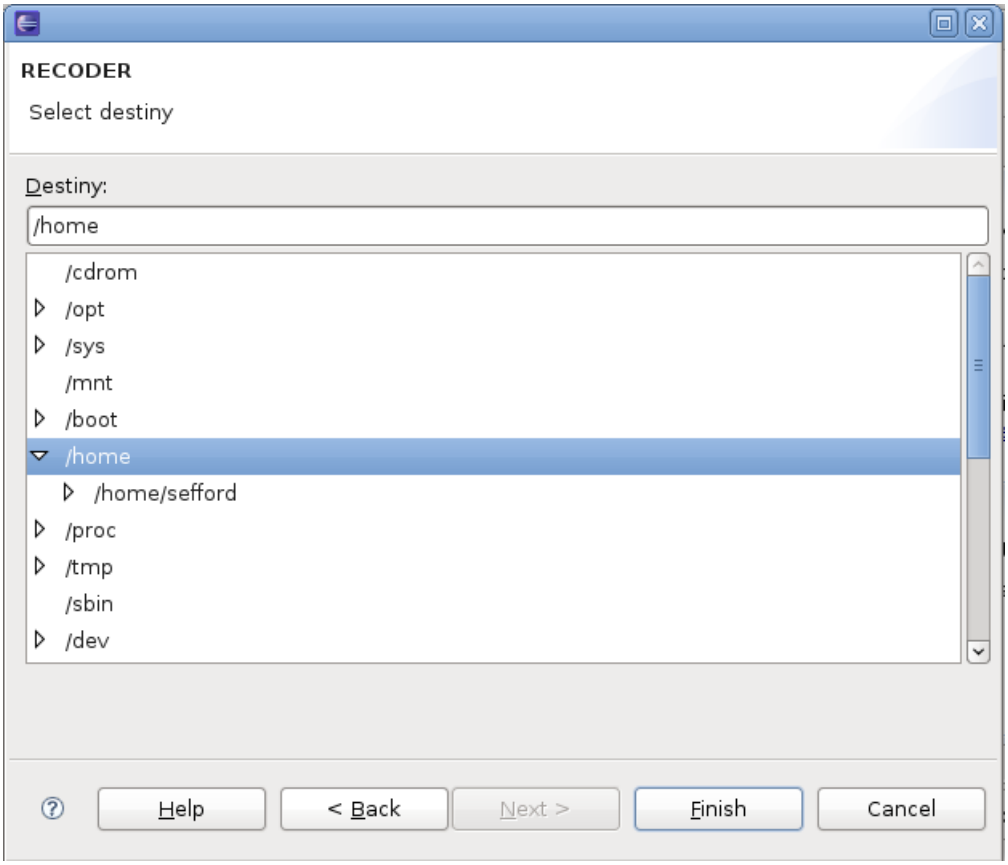

Fig. A.3: Folder selection for outputting RECODER transformation.

A.5 Results

Results are produced from those Java files configured in the selected Eclipse project. An Eclipse project can have source files or even packages excluded from the compilation. These files won't be mapped into the transformation.

The results will be outputted in the chosen folder structured as they were in the Eclipse project, providing they are not subjected to change by the transformation.

Fig. A.4: Overview of an example transformation output.

A.6 Updating RECODER plug-in

In order to update the RECODER plug-in, it is necessary to remove the previous RECODER plug-in jar files in "plugin" sub folder of Eclipse IDE installation path, and install the new version as stated in section A.1.

A.7 Uninstalling RECODER plug-in

In order to uninstall RECODER plug-in, it is necessary to remove the previous RECODER plug-in jar files in "plugin" sub folder of Eclipse IDE installation path, and restart the IDE.

Matematiska och systemtekniska institutionen
SE-351 95 Växjö

Tel. +46 (0)470 70 80 00, fax +46 (0)470 840 04
http://www.vxu.se/msi/